University
of Southern
California

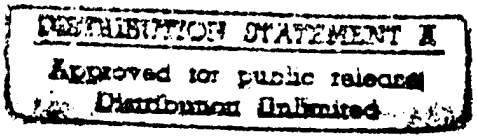
Transformational Analyses of Learning in Soar

Jihie Kim
Paul S. Rosenbloom

USC/Information Sciences Institute

January 1995

ISI/RR-95-422



INFORMATION
SCIENCES
INSTITUTE

310/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Transformational Analyses of Learning in Soar

Jihie Kim
Paul S. Rosenbloom

USC/Information Sciences Institute

January 1995

ISI/RR-95-422

REPORT DOCUMENTATION PAGE

FORM APPROVED
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Research Report

4. TITLE AND SUBTITLE

Transformational Analyses of Learning in Soar

5. FUNDING NUMBERS

N00014-92-K-2015

6. AUTHOR(S)

Jihie Kim and Paul S. Rosenbloom

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292-6695

8. PERFORMING ORGANIZATON REPORT NUMBER

ISI/RR-95-422

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, VA 22203-1714

Naval Research Laboratory
4555 Overlook Ave., SW
Washington, D.C. 20375-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

UNCLASSIFIED/UNLIMITED

12B. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This report combines two related papers, "A Transformational Analysis of Expensive Chunks" and "Mapping Explanation-Based Learning onto Soar: The Sequel."

Many learning systems must confront the problem of run time after learning being greater than run time before learning. This utility problem has been a particular focus of research in explanation-based learning (EBL). The first paper shows how the cost increase of a learned rule in chunking in Soar (a variant of EBL) can be analyzed by characterizing the learning process as a sequence of transformations from a problem solving episode to a learned rule. The analysis of how the cost changes through the transformations can be a useful tool for revealing the sources of cost increase in the learning system. Once all of the sources are revealed, by avoiding these sources, the learned rule should never be expensive.

The second paper extends the work in the first paper and the past work which analyzed chunking in Soar as a variant of EBL. The components and processes underlying EBL have been mapped to their corresponding components and processes in chunking. The paper analyzes an implementation of EBL within Soar as a sequence of transformations from a probem solving episode to a learned rule. The transformations in this sequence, along with their intermediate products, are then evaluated for their effects on the generality and expensiveness of the rules learned, and compared with the results in the first paper.

14. SUBJECT TERMS

Chunking, EBL, expensive chunks, machine learning, Soar, utility problem

15. NUMBER OF PAGES

34

16. PRICE CODE

17. SECURITY CLASSIFICTION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reoprts. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month,a nd year, if available (e.g. 1 jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the repor.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities.
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- Leave blank.
DOE	- Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	- Leave blank.
NTIS	- Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contins classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

A Transformational Analysis of Expensive Chunks

Jihie Kim and Paul S. Rosenbloom

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, U.S.A.
jihie@isi.edu, rosenbloom@isi.edu
(310) 822-1510 (x769)
Fax: (310) 823-6714

Key words: machine learning, utility problem, expensive chunks, Soar, EBL

Abstract

Many learning systems must confront the problem of run time after learning being greater than run time before learning. This utility problem has been a particular focus of research in explanation-based learning (EBL). This paper shows how the cost increase of a learned rule in an EBL system can be analyzed by characterizing the learning process as a sequence of transformations from a problem solving episode to a learned rule. The analysis of how the cost changes through the transformations can be a useful tool for revealing the sources of cost increase in the learning system. Once all of the sources are revealed, by avoiding these sources, the learned rule will never be expensive. That is, the cost of the learned rule will be bounded by the problem solving. We performed such a transformational analysis of chunking in Soar. The chunking process has been decomposed into a sequence of transformations from the problem solving to a chunk. By analyzing these transformations, we have identified a set of sources which can make the output chunk expensive.

1 Introduction

Many learning systems must confront the problem of run time after learning being greater than run time before learning. This *utility problem* has been a particular focus of research in explanation-based learning (EBL). There have been approaches which are useful for producing cheaper rules [1, 2, 3, 4, 5, 6] or filtering out expensive rules [2, 7, 8, 9]. However, these approaches cannot generally guarantee that the cost of using the learned rules will always be bounded by the cost of the problem solving from which they are learned, given the same situation. One way of finding a solution which can guarantee such cost boundness is to analyze all the sources of cost increase in the learning process and then eliminate these sources. Here we propose to approach this task by decomposing the learning process into a sequence of transformations that go from a problem solving episode, through a sequence of intermediate problem-solving/rule hybrids, to a learned rule. Analyses of these transformations then point out where extra cost is being added, and guide the proposal of alternatives that do not introduce such added costs.

The focus of the analysis in this paper is *chunking* in Soar[10]. Soar is an architecture that combines general problem solving abilities with a chunking mechanism that is a variant of explanation-based learning [11]. In the context of characterizing learning systems as a sequence of transformations, our prior work has revealed one source of added expensiveness: in chunking (and other EBL systems which use search control in the problem solving), *eliminating search control in learning can increase the cost of the learned rules* [12]. The critical consequence of the elimination of search control is that the learned rules are not constrained by the path actually taken in the problem space, and thus can perform an exponential amount of search even when the original problem-space search was highly directed (by the control rules). This analysis was based on one step (removal of search control) among the whole sequence of transformations. To reveal all sources of additional cost, we need a complete analysis of the whole sequence of transformations.

This approach is similar in spirit to [13] in its use of a transformational analysis of the learning algorithm. However, the focus of their analysis and resulting algorithm development was on speedup rather than on boundedness, and on search-control-free EBL rather than on chunking.

Section 2 of this article briefly reviews chunking in Soar. Section 3 then describes and analyzes the sequence of transformations underlying chunking. The key results of this analysis — in addition to the identification of the transformational sequence itself — are: (1) the identification of the points in the transformations at which extra cost is added; and (2) proposed modifications that may eliminate the identified sources of extra cost. Section 4 presents preliminary experimental results backing up the analysis in Section 3. Finally, Section 5 summarizes and discusses issues for future work.

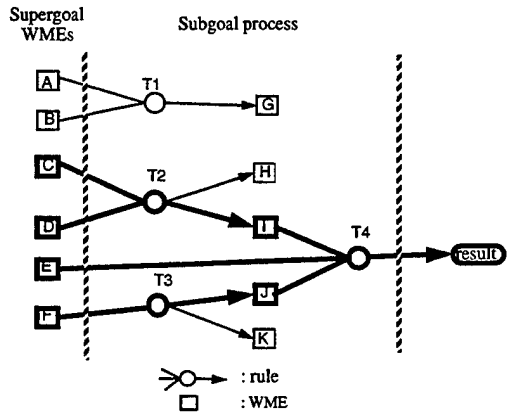


Figure 1: An example of chunking process.

2 Background

In Soar, productions comprise the *domain theory* for EBL [14, 15]. Each production consists of a set of conditions and a set of actions. Conditions test working memory for the presence or absence of patterns of tuples, where each tuple consists of an object identifier, an attribute and a value. Actions create *preferences*, each of which specifies the relative or absolute worth of a value for an attribute of a given object. Productions in Soar propose changes to working memory through these preferences, and do not actually make the changes themselves. Changes to working memory are made based on a synthesis of the preferences (by a fixed *decision procedure*). The cycle of production firings, creation of preferences, and creation of working memory elements (WMEs) underlies the problem solving.

When a situation occurs so that a unique decision cannot be made because of either incomplete or inconsistent preferences, the system reaches an *impasse*. It creates a *subgoal* to deal with the impasse. In the subgoal created for the impasse, Soar tries to resolve the impasse. Whenever a supergoal object (called a *result*) is created in the subgoal, a new chunk is created. The chunk summarizes the problem solving (rule firings) that produced the result in the subgoal.

To create chunks, Soar maintains an instantiated trace of the rules which fired in the subgoal. The *operationality criterion* in chunking is that the conditions in the chunk should be generated from the supergoal objects. By extracting the part of the trace which participated in the result creation, the system collects the supergoal (operational) elements which are connected to the result. This process is called *backtracing*, and the instantiated trace is called a *backtrace*. It corresponds to the *proof tree* (or *explanation*) in EBL. The resulting supergoal elements are variabilized and reordered by a heuristic algorithm, and become the conditions of the chunk. The action of the chunk is the variabilization of the result. An example of chunking is shown schematically in Figure 1. The two striped vertical bars mark the beginning and the

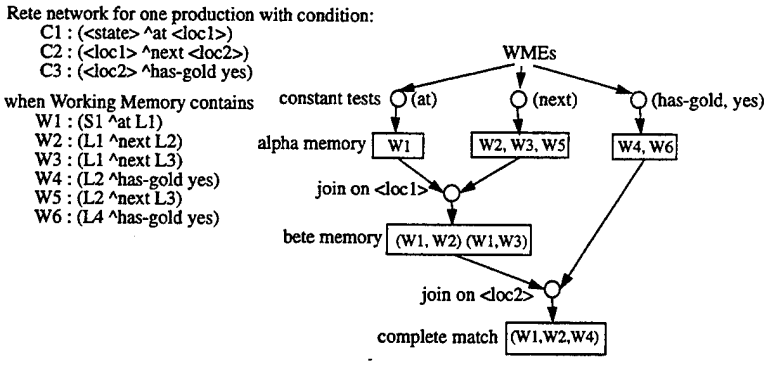


Figure 2: Rete network of a rule.

end of the subgoal. The WMEs to the left of the first bar exist in the supergoal (prior to the creation of the subgoal). The objects (WMEs) between the two bars are internal to the subgoal. The object to the right of the second bar is the result of the subgoal. T1, T2, T3 and T4 are traces of the rule firings. For example, T1 records a rule firing which examined WMEs A and B and generated a preference suggesting WME G. The highlighted rule traces are those included in the backtrace; T2, T3, and T3 have participated in the result creation.

The chunking process can also be characterized in a different way — instead of simply considering it as a procedure which has problem solving episodes as input and learned rules (chunks) as output, it can be considered as a *sequence of transformations* from problem solving episodes, through intermediate *pseudo-chunks*, to chunks. The cost changes through the transformations can be estimated by analyzing each step.

Note that when we compare the cost of a problem solving episode to the cost of a chunk, by "cost" we will mean just the match cost of all of the rules that fired to generate the result (whether this be via multiple rules during the initial problem solving, or via a single chunk).[1] Because computing match cost is dependent on the match algorithm used, we briefly review the Rete algorithm [16] employed in Soar.

Rete is one of the most efficient rule-match algorithms presently known. Its efficiency stems primarily from two key optimizations: *sharing* and *state saving*. Sharing of common conditions in a production, or across a set of productions, reduces the number of tests performed during match. State saving preserves the previous (partial) matches for use in the future. Figure 2 illustrates a Rete network for a rule. Each WME consists of an object identifier, an attribute (indicated by an up-arrow(^)), and a value. Symbols enclosed

[1]Actually, the cost of a problem solving episode also includes the costs of firing rules and of making decisions. However, we will not explicitly focus on these factors here because they drop out during the transformational process.

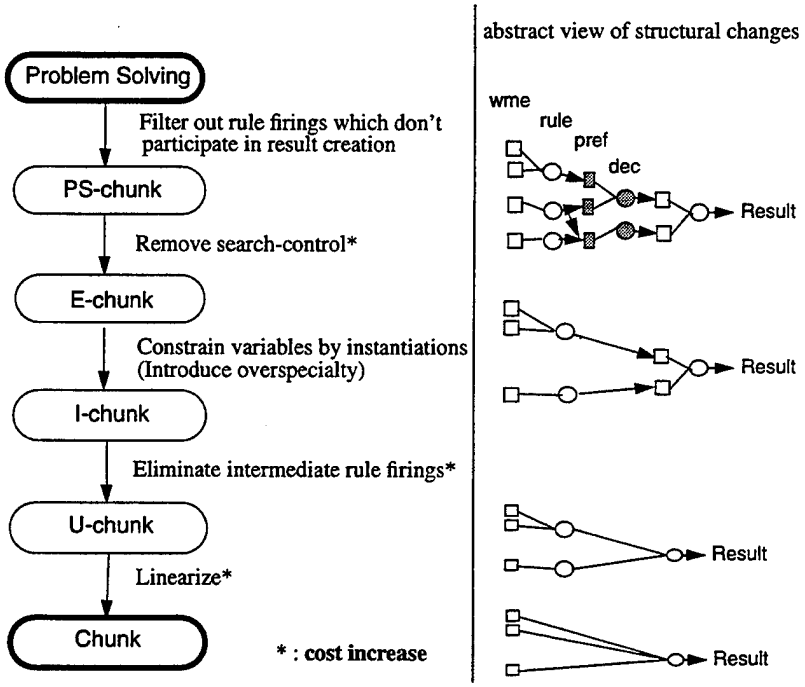


Figure 3: A sequence of transformations from a problem solving to a chunk.

in angle brackets are variables. The conditions of the rule are compiled into a data flow network. The network has two parts. The alpha part performs constant tests on WMEs, such as tests for **at** and **yes**. The output of these tests are stored in alpha memories. Each alpha memory contains the set of WMEs which pass all of the constant tests of a condition (or more than one, if it is shared). The beta part of the network contains join nodes and beta memories.[2] Join nodes perform consistency tests on variables shared between conditions, such as <loc1>, which is shared between C1 and C2. Beta memories store partial instantiations of productions, that is, instantiations of initial subsequences of conditions. The partial instantiations are called *tokens*. Because match time per token is known to be approximately constant in Rete [17, 6] — and because counting tokens yields a measure that is independent of machines, optimizations, and implementation details — we will follow the standard practice established within the match-algorithm community and use the number of tokens, rather than time, as our comparative measure of match cost.

3 Transforming problem solving to a chunk

Figure 3 shows the sequence of transformations that convert a problem solv-

[2]There also are negative nodes, into which negative conditions are compiled. A negative node passes a partial instantiation when there are no consistent WMEs.

```
R1)
1 (goal <g1> ^super-goal <g2>)
2 (goal <g2> ^impasse-object <q>)
 -->
(<g1> ^eval-operator <o> CAND)
(<o> ^evaluate <q>)
; if <g2> is a subgoal of <g1> because of
; an impasse-object <q>, then create
: a candidate evaluation-operator <o>
; for evaluating <q>

R2)
2 (goal <g> ^eval-operator<o> CAND)
2 (<o> ^evaluate <q>)
1 (<q> ^priority 1)
-->
(<g> ^eval-operator <o> BEST)
; if <o> is a candidate evaluation-operator
; for evaluating <q>, and <q> has priority 1
; then try to evaluate <o> first by making it
; the best alternative among the candidates.

R3)
1 (goal <g> ^eval-operator <o>)
1 (<o> ^evaluate <q2>)
1 (<q2> ^transportation <x>)
2 (<q2> ^goto <l1>)
2 (<l1> ^next-to goal)
-->
(<o> ^evaluated-as success)
; if <o> is the evaluation-operator for evaluating
; <q2> and <q2> goes to a position next to the goal
; by some transportation, then <o> is evaluated as success

R4)
S (goal <g1> ^eval-operator <o>)
S (<o> evaluate <q>)
1 (goal <g1> ^super-goal <g3>)
1 (<o> ^evaluated-as success)
-->
(<g2> ^object <q> BEST)
; if <o> is the evaluation-operator for evaluating
; <q> and <o> is evaluated as success,
; resolve the impasse by making <q> best
```

```
Given WMEs
    W1 : (G2 ^super-goal G1)
    W2 : (G1 ^impasse-object Q1)
    W3 : (G1 ^impasse-object Q2)
    W4 : (Q1 ^transportation car)
    W5 : (Q2 ^transportation train)
    W6 : (Q2 ^transportation car)
    W7 : (Q2 ^transportation bus)
    W8 : (Q1 ^priority 1)
    W9 : (Q2 ^priority 2)
    W10: (Q1 ^goto L1)
    W11: (Q1 ^goto L2)
    W12: (Q2 ^goto L3)
    W13: (Q2 ^goto L4)
    W14: (L1 ^next-to goal)
    W15: (L2 ^next-to goal)

Created WMEs and preferences during
problem solving
    P16: (G2 ^eval-operator O1 CAND)
    P17: (G2 ^eval-operator O2 CAND)
    W18: (O1 ^evaluate Q1)
    W19: (O2 ^evaluate Q2)
    P20 : (G2 ^eval-operator O1 BEST)
    W21: (G1 ^eval-operator O1)
    W22: (O1 ^evaluated-as success)
    P23: (G1 ^object Q1 BEST)
```

Figure 4: An example Soar task.

ing episode into a chunk. Each transformation (except for the last) creates an intermediate structure, called a *pseudo-chunk*. As the sequence progresses, the pseudo-chunks become more like chunks and less like problem solving. Each pseudo-chunk can itself be matched and fired (given an appropriate interpreter) and thus independently create the result. The cost of a pseudo-chunk can be determined by counting the number of tokens generated during the match to produce the result. By analyzing how the transformations alter these costs, the sources of added expensiveness are revealed.

The following subsections discuss each transformation shown in Figure 3, including their resulting (pseudo-)chunks and their effects on cost. These discussions are presented in the context of a simple illustrative task — that of evaluating which mode of transportation is best in particular situations (Figure 4). There are four rules and fifteen WMEs to begin with in this task. In the figure, a number in front of a rule condition denotes the number

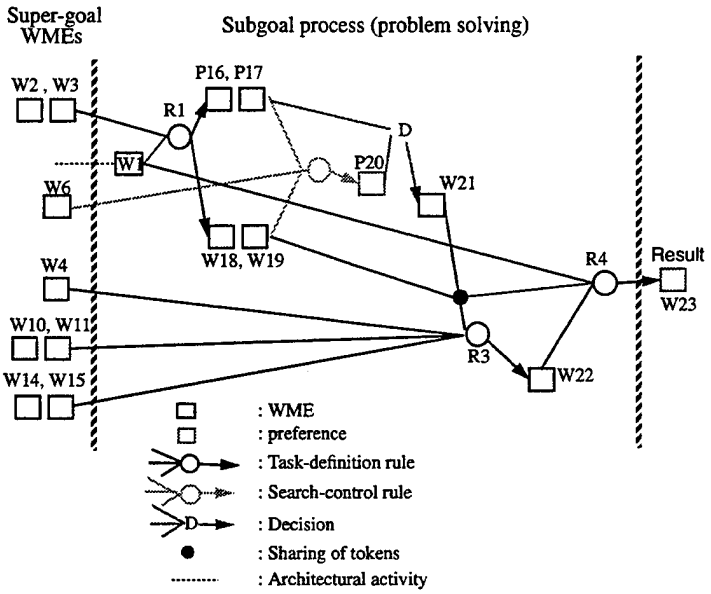


Figure 5: Problem solving episode excluding unnecessary rule firings. This structure embodies PS-chunk.

of tokens generated by in the problem solving episode shown in Figure 5 by joining the tokens passed on from the previous conditions with the WMEs in the condition's alpha memory. Figure 5 shows how the sequence of rule firings during the problem solving episode creates the result (G1 ^object Q1 BEST)[3], given the rules and the WMEs. A connection from one rule to another rule through a decision means that preferences created by the former rule participated in the decision for the WME which is matched to a condition of the latter rule. The trivial decision steps — creation of one candidate and the following creation of a WME from the candidate — are not shown for brevity. Actual problem solving normally includes other rule firings which are not linked to the result creation; however, those are omitted here.

3.1 Filtering out unnecessary rule firings (⇒ PS-chunk)

As a first step toward producing a chunk, we can filter out the unnecessary rule firings which did not participate in the result creation. For the given example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 5. The resulting pseudo-chunk — called a *PS-chunk* (Problem-Solving-like chunk) — looks very similar to the original problem solving, aside from the missing unnecessary parts.[4] However, its

[3]This preference means that Q1 is the best alternative among the candidate values, given the identifier G1 and the attribute object.

[4]In addition, architectural actions that occurred during the problem solving episode are replaced in the PS-rule by dummy rules that have the same effect, much in the way that architectural axioms are used in Prodigy/EBL[2].

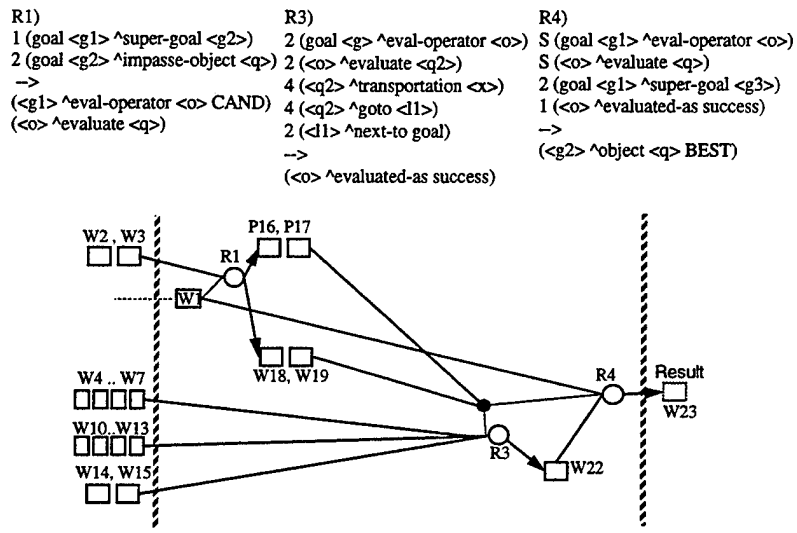


Figure 6: E-chunk: results from eliminating search control in the PS-chunk.

processing differs significantly from the initial problem solving by being closed off from intermediate WMEs generated outside of this structure.[5] For example, the link between R3 and R4 through W22 means that no other WMEs except for those created by R3 are matched to the condition of R4. The only parts of a PS-chunk that are exposed to the full set of WMEs are the conditions matched to the supergoal elements, and the result creation. The key difference between a PS-chunk and a normal chunk is that matching a PS-chunk requires replaying (part of) problem solving, while matching a normal chunk requires just one rule match. Either can create the result in a similar circumstance.

The cost (number of tokens) of a PS-chunk is bounded by the cost of problem solving. If there were unnecessary rule firings in the problem solving (as is usually the case), the cost of a PS-chunk is strictly less than the cost of the problem solving. If not, the cost is the same as the problem solving.

3.2 Removing search control (⇒ E-chunk)

PS-chunks contain all rules involved in the result creation. However, chunking employs only traces from *task-definition rules*; that is, rules that directly propose values of WMEs. *Search-control rules*, as distinguished from task-definition rules, suggest the relative worth of the proposed values. The search-control rules are missing in chunking [18, 10] (and other EBL systems [19]) based on the assumption that they only affect the efficiency, not the correctness of learned rules. This omission is intended to increase the generality of the learned rules — reducing the number of conditions by leaving out search control rules means less restriction on the test of applicability of the rules,

[5]It is not different in how it uses such optimizations as sharing and state saving; for example, the tokens from the first two conditions of R4 are shared with the tokens from the first two conditions of R3.

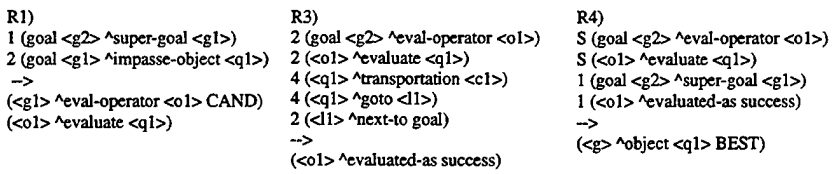


Figure 7: I-chunk: created by constraining variables (by instantiations) in an E-chunk. The structure remains the same as in the E-chunk (Figure 6) for this example.

and thus implies increased generality. An *E-chunk* (Explanation-structure-like chunk) is the intermediate structure which is formed by removing search control from a PS-chunk. Figure 6 shows the E-chunk created from the PS-chunk in Figure 5. The search-control rule R2 is gone, and all proposed candidates become WMEs without filtering through the search control in the decision process. This structure can be mapped onto the normal backtrace in chunking. The only difference between an E-chunk and a backtrace is that a backtrace consists of instantiations while an E-chunk consists of rules. By replacing the rules in the trace, a backtrace can be directly mapped to an E-chunk. An E-chunk is similar to an EBL explanation structure.

The consequence of eliminating search control is that the E-chunk is not constrained by the path actually taken in the problem space, and thus can perform an exponential amount of search even when the original problem-space search was highly directed (by the control rules), as analyzed in [12]. In the above example, without constraining eval-operator to the best candidate — which has priority 1 — the number of tokens in the match of rule R3 increases from 7 to 14. Overall, the total number of tokens increases from 17 to 20. This is thus one of the star-marked (i.e., cost increasing) transformations in Figure 3.

One promising way of avoiding this problem is to *incorporate search control in chunking*[12]. By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

3.3 Constraining variables by instantiations(⇒I-chunk)

The variabilization step in chunking is performed by examining the backtrace (explanation). All constants are left alone; they are never replaced by variables. All object identifiers in the instantiations are replaced by variables; and in particular, all occurrences of the same identifiers are replaced by the same variable. Since E-chunks consist of rules rather than instantiations, we need to model chunking's variabilization step as the strengthening of constraints on the match rather than as the weakening of them. If a variable is instantiated as a constant, it is replaced by that constant. If a variable is instantiated by an identifier, it remains as a variable, but may possibly

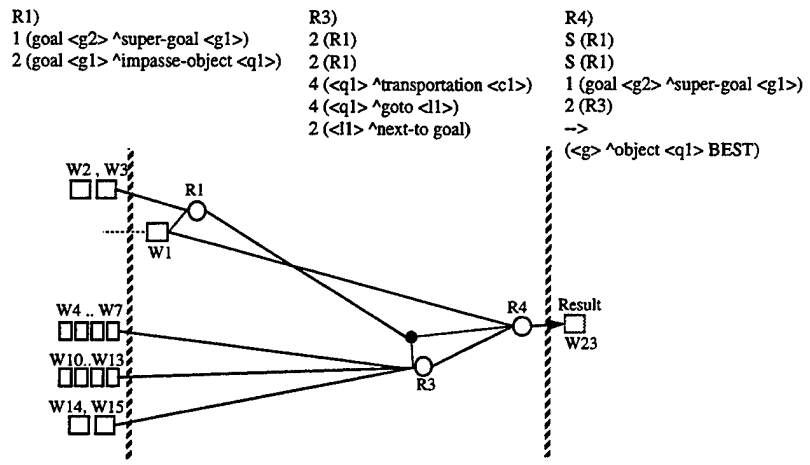


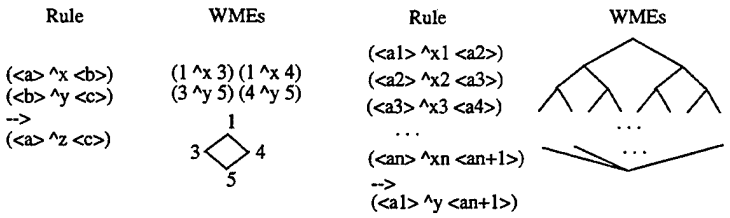
Figure 8: U-chunk: results from eliminating intermediate rule firings in the I-chunk.

undergo a name change; in particular, all occurrences of variables which are instantiated by the same identifier are replaced by the same variable. For example, the variables in Figure 6 can be constrained as shown in Figure 7. The pseudo-chunk generated by this step is called an *I-chunk* (instantiation-based chunk).

This transformation can *overspecialize* learned rules when distinct variables in the original rules accidentally happen to match the same identifier; for example, although variable <g2> in R1 and variable <g3> in R4 (Figure 6) is instantiated by same the identifier G1, and changed to the same variable <g1>, they can correctly be generalized as different variables. However, from the perspective of cost, this transformation doesn't increase the number of tokens. The number of tokens generated should remain the same, or be reduced by the introduced constraints.

3.4 Eliminating intermediate rule firings (⇒ U-chunk)

This step unifies the separate rules in an I-chunk into a single rule, called a *U-chunk* (unified chunk). Figure 8 shows the result of unifying the example problem I-chunk into the corresponding U-chunk. Although R1, R3, and R4 still have their own identifiable conditions in the U-chunk, there are now no intermediate rule firings. The boundaries between the rules are eliminated by removing the intermediate processes of rule firing and WME creation. In lieu of these processes, the instantiations generated by matching the earlier rules in the firing sequence (i.e., the tokens produced by their final conditions) are passed directly to the match of the later rules. In effect, this step replaces the intermediate WMEs with the instantiations which created the WMEs. For example, one of R4's conditions receives the instantiations of R3 directly as intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1, R3, and R4 are no longer (separate) rules.



(a) An example case of increased tokens (b) A potential worst case for U-match

Figure 9: Number of tokens can increase in a U-chunk.

To match U-chunks, an extension is required to the Rete algorithm. The traditional form of the algorithm, as shown in Figure 2, requires a *linear* match network, in the sense that a total ordering must be imposed on the conditions to be matched; such as C1, then C2, and then C3. In (linear) Rete, each join node checks the consistency of a token (a partial instantiation) and a WME, with each token itself being a sequence of WMEs, each of which matches one condition. However, U-chunks require the ability to perform *non-linear* matches, in which conditions are matched hierarchically via join nodes that compare pairs of tokens, rather than just a single token and a WME. They also require the ability to create hierarchically structured tokens (when pairs of incoming tokens are consistent); that is, a token must now be a sequence of WMEs or tokens (instantiations of a rule).

One benefit of going with U-chunks, rather than I-chunks, is that they enable equality tests across sub-structures which previously represented separate rules. For example, we can now test equality between the instantiations of <q1> in R1 and R3. However, cost problems are introduced in going to U-chunks because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations. For example, given the rule and WMEs in Figure 9-(a), two instantiations — (1 ^x 3) (3 ^y 5) and (1 ^x 4) (4 ^y 5) — are created. Because these two instantiations generate the same bindings for variables <a> and <c>, only one tuple (WME) is generated in the problem solving.[6] In this case, the number of tokens is increased after we replace the WMEs with the instantiations. This really happens in our example. While the two instantiations of R3 are collapsed into one WME and supplied to the fourth condition of R4 in the I-chunk, the two instantiations are directly used in the U-chunk, and create one more token. A worst case can arise when the working memory is structured as in Figure 9-(b). While the number of instantiations is exponential in the number of conditions, the number of WMEs is only one.

Our proposed solution to this problem is to *preprocess instantiations before they are used* so that the number of tokens passed from a substructure of a U-chunk is no greater than the number of WMEs passed in the corresponding

[6]Working memory is a set in Soar (and other Ops-like languages), and does not include duplicate elements.

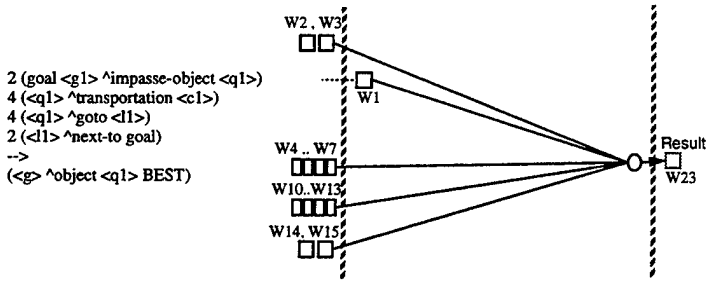


Figure 10: Chunk: results from linearizing the U-chunk.

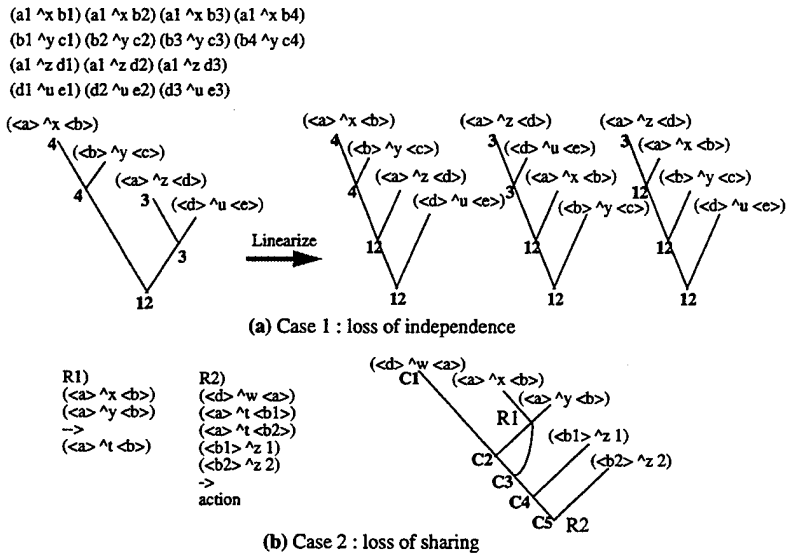


Figure 11: Linearization can increase the number of tokens.

I-chunk. This could potentially be done either be grouping instantiations that generate the same WME or by selecting one of them as a representative.

3.5 Linearizing (⇒ Chunk)

A U-chunk can be linearized to become a chunk. The hierarchical structure of U-chunks is flattened into a single layer, and the conditions are totally ordered. For example, the non-linear structure in Figure 8 can be flattened to the structure in Figure 10. After the flattening, chunking uses a heuristic condition-ordering algorithm to further optimize the resulting match.

The linearization transformation turns out to introduce three ways in which match costs can increase. The first way arises directly from the flattening of the U-chunk's hierarchical structure. In a U-chunk, the conditions in a sub-hierarchy (e.g. the conditions in R1) are matched independently from the other parts of the structure before its created instantiations are joined with the others. By combining these sub-hierarchies together — through linearization — some of the previously independent conditions get joined with other parts

	Number of tokens
Problem Solving	52
PS-chunk	42
E-chunk	108
I-chunk	108
U-chunk	198
L-chunk	215

Table 1: Number of tokens of each step in a Grid Task.

of the structure before they finish their sub-hierarchy match. This change can increase the number of tokens. For example, after linearizing the U-chunk in Figure 11-(a), the number of tokens increases no matter what condition ordering is used. In the worst case, the increase can be exponential in the number of hierarchical levels in the U-chunk.

The second way arises from the impact flattening can have on sharing. As long as the implementation of Rete cannot capture the sharing from the non-linear structure (of the U-chunk), the number of tokens can increase. For example, in Figure 11-(b), sharing of sub-tokens from R1 for C2 and C3 in R2 cannot be realized in a linearized structure.

The third way arises because the heuristic condition-ordering algorithm cannot guarantee optimal orderings. Whenever this algorithm creates a non-optimal ordering, additional cost may be occurred.

Our proposed solution to this set of problems is to *eliminate the linearization step.* By keeping the hierarchical structure — that is, by replacing chunks with U-chunks — all three causes of cost increase can be avoided. The key thing that this requires is an efficient generalization of Rete for non-linear match.

4 Experimental Results

In order to supplement the abstract analysis provided in the previous section with experimental evidence, we have implemented a set of learning algorithms that correspond to the set of initial subsequences of the overall transformation sequence; that is, each learning algorithm in the set starts with the problem solving episode and generates a distinct type of (pseudo-)chunk. We have also implemented the extensions to the Rete algorithm necessary to allow all of the types of pseudo-chunks to match and fire. At each stage from problem solving to chunks, match cost is evaluated by counting the number of tokens required during the match to generate the result.

So far, the resulting experimental system has been applied to a simple Grid-task problem[6] which creates one subgoal to break a tie (impasse) among the candidate operators and creates a chunk. The results of this experiment are shown in table 1. The pattern of cost increases matches the expectations

generated from the earlier analysis in that a transformation led to increased cost on this task if and only if it was identified by the analysis as a cost increasing transformation.

5 Summary and Future Work

We have performed an analysis of the chunking process as a sequence of transformations from a problem solving episode to a chunk. By analyzing these transformations, we have identified a set of sources which can make the output chunk expensive. We conjecture there are no other sources of cost increase, but cannot yet prove this.

Based on the above analysis and the proposed potential solutions to the sources of expensiveness, we are currently working towards the specification and implementation of a variant of chunking which does not introduce any of these sources. If it works, the cost of using a chunk should always be bounded by the cost of the corresponding problem solving.

A similar transformational analysis can also be performed for EBL. As with the analysis of chunking, this analysis should identify sources of expensiveness in EBL, and help guide the design of safer EBL mechanisms. In addition, a parallel analysis of EBL and chunking should further clarify the relationship between the two. An earlier comparison related the four basic structural components (goal concept, domain theory, training example, operationality criterion) of the two systems[11]; however, a transformational analysis should allow us to go beyond this to a deeper analysis of the processes underlying the two algorithms. A preliminary analysis of EBL shows that the sequence of transformations underlying EBL is very similar to that in chunking, except for the regression process, where chunking uses instantiations. In addition, EBL systems seem to suffer from cost problems similar to those that show up for chunking.

Acknowledgments

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan as part of contract N00014-92-K-2015 from the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). We would like to thank Milind Tambe and Bob Doorenbos for helpful comments on this work.

References

[1] A. E. Prieditis and J. Mostow. Prolearn: Towards a prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 494–498, 1987.

[2] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, 1988.

[3] P. Shell and J. Carbonell. Empirical and analytical performance of iterative operators. In *The 13th Annual Conference of The Cognitive Science Society*, pages 898–902. Lawrence Erlbaum Associates, 1991.

[4] Jude W. Shavlik. Aquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70, 1990.

[5] O. Etzioni. Why prodigy/ebl works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 916–922, 1990.

[6] M. Tambe. *Eliminating combinatorics from production match.* PhD thesis, Carnegie-Mellon University, 1991.

[7] R. Greiner and I. Jurisica. A statistical approach to solving the ebl utility problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 241–248, 1992.

[8] J. Gratch and G. Dejong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Aritificial Intelligence*, pages 235–240, 1992.

[9] S. Markovitch and P. D. Scott. Information filtering : Selection mechanism in learning systems. *Machine Learning*, 10(2):113–151, 1993.

[10] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325, 1991.

[11] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, 1986. AAAI.

[12] J. Kim and P. S. Rosenbloom. Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 174–181, 1993.

[13] A. Segre and C. Elkan. A high-performance explanation-based learning algorithm. *Artificial Intelligence*, 69:1–50, 1994.

[14] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization – a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[15] G. F. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[16] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[17] M. Tambe, D. Kalp, A. Gupta, C. L. Forgy, B. G. Milnes, and A. Newell. Soar/psm-e: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, 1988.

[18] J. E. Laird, P. S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57, 1986.

[19] S. Minton. Personal communication. 1993.

Mapping Explanation-Based Learning onto Soar: The Sequel

Jihie Kim and Paul S. Rosenbloom

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, U.S.A.
jihie@isi.edu, rosenbloom@isi.edu

Key words: EBL, chunking, Soar, utility problem

Abstract

In past work, chunking in Soar has been analyzed as a variant of explanation-based learning. The components and processes underlying EBL have been mapped to their corresponding components and processes in chunking. The cost and generality of the resulting rules have also been compared. Here we extend that work by analyzing an implementation of EBL within Soar as a sequence of transformations from a problem solving episode to a learned rule. The transformations in this sequence, along with their intermediate products, are then evaluated for their effects on the generality and expensiveness of the rules learned, and compared with the results of a similar analysis previously performed for chunking. The analysis reveals that EBL, as implemented for Soar, yields the same sources of expensiveness and overgenerality as does chunking — and that, in fact, these problems stem more from other aspects of Soar than from the details of either learning algorithm. (However, some of these aspects may appear in other AI architectures, even though the analysis is based on Soar.) Moreover this analysis reveals that the differences between EBL and chunking are localized within a single transformation, where chunking overspecializes with respect to EBL.

1 Introduction

Explanation-based learning (EBL)[1, 2] can improve the performance of problem solving systems by learning new rules. Given the four informational components (the goal concept, the training example, the domain theory, and the operationality criterion), EBL generates a new rule through three steps: (1) use the domain theory to prove that the training example is an instance of the goal concept; (2) create an explanation structure from the proof, filtering out irrelevant rules and facts; and (3) regress the goal concept through the explanation structure (until it reaches operational predicates), and create general conditions under which the explanation structure is valid.

In past work, chunking in Soar[3] has been analyzed as a variant of EBL. The four components and the three steps of EBL have been mapped to the components of Soar and to the sub-processes of chunking, respectively [4]. Also, the cost and the generality of the learned rules have been compared [5]. We have recently extended this earlier work by implementing EBL within Soar (Version 6) — to yield *EBL/Soar* — and then analyzing this implementation as a sequence of transformations from a problem solving episode to a learned rule. Each transformation (except for the last) yields an executable intermediate structure called a *pseudo-rule*, which can be evaluated with respect to both its generality and cost, and thus directly compared in terms of generality and cost against both the original problem solving and the ultimate *EBL-rule*.

The results of this analysis are then compared with a similar transformational analysis of chunking. This analysis comparison reveals that EBL, as implemented for Soar, yields the same sources of expensiveness and overgenerality as does chunking — and that, in fact, these problems stem more from other aspects of Soar than from the details of either learning algorithm. This comparison also reveals that the differences between EBL and chunking are localized within a single transformation, where chunking overspecializes with respect to EBL. A set of modifications to Soar that have recently been proposed to eliminate the possibility of a chunk being more expensive than the problem solving episode from which it was learned, thus look like they may also be sufficient to provide the same guarantees for EBL.

2 Chunking and EBL/Soar

In Soar, productions comprise the *domain theory* for EBL. Each production consists of a set of conditions and a set of actions. Conditions test working

memory for the presence or absence of patterns of tuples, where each tuple consists of an object identifier, an attribute and a value. Actions create *preferences*, which specify the relative or absolute worth of values for attributes of objects. Productions in Soar propose changes to working memory through these preferences, but do not actually make the changes themselves. Changes to working memory are based on a synthesis of the preferences by a fixed *decision procedure*. When a situation occurs in which a unique decision cannot be made because of either incomplete or inconsistent preferences, Soar reaches an *impasse*, and creates a *subgoal* in which it tries to resolve the impasse. Whenever a supergoal object (called a *result*) is created in the subgoal, chunking creates a new rule (called a *chunk*). The chunk summarizes the problem solving (rule firings) that produced the result. EBL/Soar can use the same problem solving episode as input for creating a new rule. Its *operationality criterion* is that the conditions should be generated from the supergoal objects. The *training example* is the supergoal situation. The *goal concept* is simply the result.

To assist chunking, Soar maintains an instantiated trace of the rules which fired in the subgoal. By extracting the part of the trace which participated in the result creation, chunking collects the supergoal (operational) elements which are connected to the result. This process is called *backtracing*, and the instantiated trace is called a *backtrace*. This backtrace can act as the proof tree (or explanation) in EBL/Soar. Given the backtrace, chunking variabilizes and reorders the resulting supergoal elements, and creates the conditions of the chunk. The action of the chunk is the variabilization of the result. Given the backtrace (explanation), EBL/Soar can construct an explanation structure by replacing the instantiations with the general rules. Application of the regression process to the explanation structure can yield a set of general conditions under which the explanation structure is valid.

The cup domain, a typical illustrative EBL task, can be represented by the Soar rules shown in Figure 1-(a). The training example (i.e., the supergoal situation) is shown in Figure 1-(b), as the WMEs that existed before the subgoal process. Each WME is represented by a tuple that contains three items: object identifier, attribute (up-arrow(^) indicates attribute name), and value. Symbols enclosed in angle brackets are variables. R1 can create a new problem-space named "cup" and a new state, given the lack of information in the supergoal situation about which object is a cup. In R2, R3, and R4, the training example is accessed through the attribute **super-state** which links the cup problem-space state and the supergoal state.

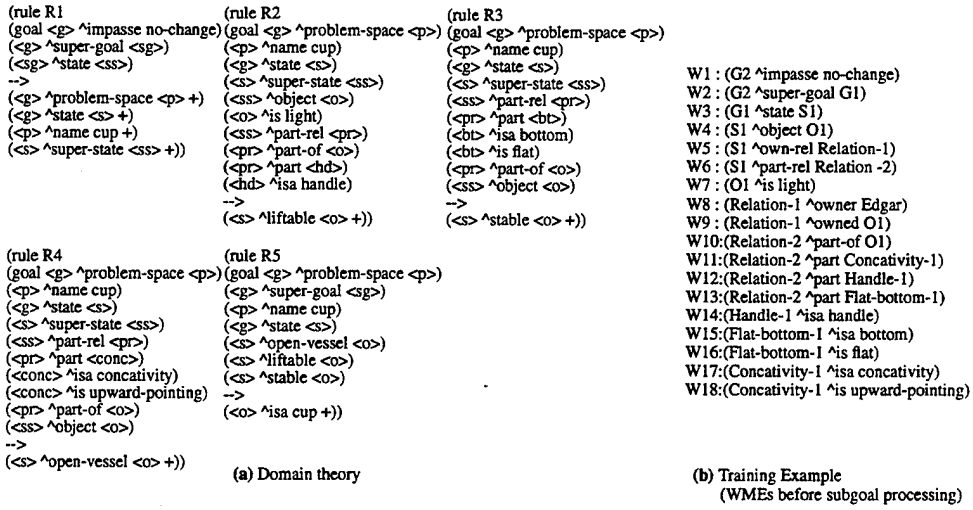


Figure 1: Cup domain as a Soar task.

3 Transforming problem solving to a rule

Figure 2-(a) shows the two sequences of transformations that represent chunking and EBL/Soar. (The chunking part is adapted from [6]). Each transformation (except for the last) creates an intermediate structure which is called a *pseudo-chunk* in chunking and *pseudo-rule* in EBL/Soar. As the sequences progress, the pseudo-chunks (or pseudo-rules) become more like chunks (or EBL/Soar rules) and less like problem solving. Each pseudo-chunk (or pseudo-rule) can itself be matched and fired (given an appropriate interpreter) and thus independently create the result. By comparing the two sequences, we can clarify the relationship between the two systems. Also, by analyzing how the transformations alter cost and generality, a set of sources of added expensiveness and changes in generality can be contrasted.

Note that when we compare the cost of a problem solving episode to the cost of a (pseudo-) rule, by "cost" we will mean just the match cost of all of the rules that fired to generate the result (whether this be via multiple rules during the initial problem solving, or via a single learned rule).[1] Soar employs Rete, one of the most efficient rule-match algorithms presently known, as the match algorithm. Its efficiency stems primarily from two key optimizations: *sharing* and *state saving*. Sharing of common conditions in a production, or across a set of productions, reduces the number of tests performed during

[1]The total cost of a problem solving episode also includes the costs of firing rules and of making decisions. However, we will not explicitly focus on these factors here because they drop out during the transformational process.

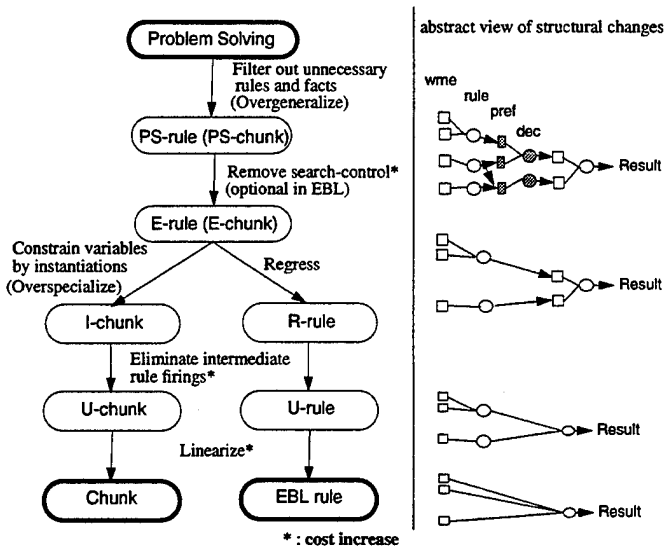


Figure 2: The transformational sequences underlying chunking and EBL/Soar.

match. State saving preserves the previous (partial) matches for use in the future. A partial instantiation of a rule, also called a *token*, is a consistent binding of variables in a subset of the conditions. Because match time per token is known to be approximately constant in Rete[7, 8], we use the number of tokens as a tool for measuring the complexity.[2]

The following subsections analyze the transformations underlying EBL/Soar, along with their resulting (pseudo-) rules and their effects on cost and generality. This analysis is then compared with the results from analyzing the corresponding transformations and intermediate results in chunking. Examples are taken from the cup domain (Figure 1).

3.1 Filtering out unnecessary rule firings (⇒PS-rule)

The **Problem Solving** node in Figure 2 represents the problem solving episode in Soar. Its generation via problem solving corresponds to the EBL step of "using the domain theory to prove that the training example is an instance of the goal concept". The first transformation applies to this episode, and filters out any rule firings which did not participate in creating the result. In the cup example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 3. The resulting pseudo-rule — called a *PS-rule* (Problem-Solving-like rule) — looks very similar

[2]Counting tokens is the standard practice within the match-algorithm community because it is independent of machines, optimizations, and implementation details.

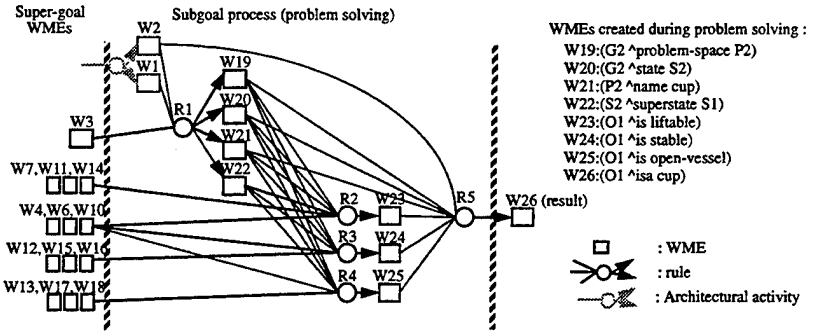


Figure 3: Problem solving episode excluding unnecessary rule firings. This structure embodies both a PS-rule and a PS-chunk.

to the original problem solving episode, aside from the missing unnecessary parts. Also, its implementation incorporates the same optimizations that are used in the original problem solving; for example, the tokens from the first four conditions of R3 and R4 are still shared with the tokens from the first four conditions of R2 in the match network for the PS-chunk. However, its execution differs significantly from the initial problem solving episode in being closed off from intermediate WMEs generated outside of this structure. For example, the link between R2 and R5 through W23 means that no other WMEs except for those created by R2 are matched to the condition of R5. The only parts of a PS-rule that are exposed to the full set of WMEs are the conditions matched to the supergoal elements. The key difference between a PS-rule and a EBL rule is that matching a PS-rule requires replaying (part of) problem solving, while matching a EBL rule requires just one rule match. Either can create the result in a similar circumstance. *PS-chunks* are the same as PS-rules. As a first step, chunking also filters out unnecessary rule firings in the given problem solving.

In Soar, some problem solving activities do not involve rule firings. For example, the acts of signaling that an impasse has occurred and creating a subgoal are performed by the architecture itself, not by the firing of rules. Ignoring these activities can leave holes in the backtrace. So Soar implicitly provides two architectural axioms that model these architectural actions, much as in [9]. First, if a (architecture created) WME is obviously based on a supergoal object, a dummy instantiation that links them is created and added to the backtrace. Second, if it is intractable to compute the linkage to supergoal objects, the backtrace simply ignores the architectural WME — just as if it had been created by a rule with no conditions. This may yield overgeneralization, but in return it helps maintain tractability. The use of these architectural axioms is driven by the nature of Soar's architectural

actions, and is independent of whether learning occurs via chunking or EBL. Thus PS-rules and PS-chunks share the possible source of overgenerality.

Because PS-rules are created by filtering out unnecessary rules in problem solving, and their implementation preserves match optimizations, the cost (number of tokens) of a PS-rule (and a PS-chunk) is bounded by the cost of problem solving. If there were unnecessary rule firings in the problem solving (as is usually the case), the cost of a PS-rule is strictly less than the cost of the corresponding problem solving. If not, the cost is identical.

3.2 Removing search control ($\Rightarrow$ E-rule)

PS-rules incorporate all rules which are linked to the result creation. That is, they include not only *task-definition rules* (rules that directly propose values of WMEs), but also *search-control rules* (rules that suggest the relative worth of the proposed values). However, in archetypical EBL systems implemented for Prolog-like languages, the problem solving does not employ search-control rules. Even in Prodigy/EBL, where the problem solving involves search control, the explanation ignores search control rules [10].

An *E-rule* (Explanation-structure-like rule) is the intermediate structure which is formed by removing search control (if there is any) from a PS-rule. Because no search control is used in the cup domain, the structure of the E-rule is the same as the PS-rule shown in Figure 3. The E-rule acts as an EBL explanation structure.

The search-control rules are also missing in chunking[11, 3], based on the assumption that they only affect efficiency, and not correctness of learned rules. The intended purpose of this omission is to increase the generality of the learned rules, by reducing the number of conditions incorporated into learned rules. Given that the PS-rule is the same as the PS-chunk, and both are transformed in the same way, the *E-chunk* is the same as the E-rule.

Unfortunately, the consequence of eliminating search control (if there was any) is that the E-rule is not constrained by the path actually taken in the problem space, and thus can perform an exponential amount of search even when the original problem-space search was highly directed (by the control rules), as analyzed in [12]. One possible way of avoiding the problem is to *incorporate search control into the explanation structure*, so that the match process for the learned rule focuses on only the path that was actually followed. This can specialize the learned rule, but in return it enables the rule's cost to remain bounded by the cost of the original problem solving [12].

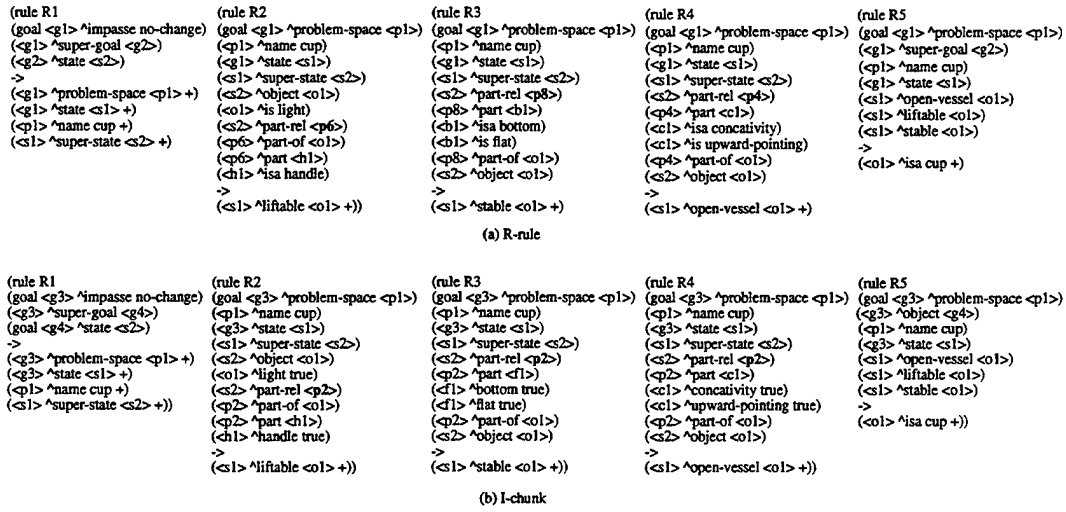


Figure 4: (a) R-rule: created by regressing the E-rule; (b) I-chunk: created by constraining variables by instantiations. The structure remains the same as in the E-rule (E-chunk) for this example.

3.3 Regress (⇒R-rule)

We can apply the regression process of EBL [13] to the E-rule. Replacing the variable names with unique names and then unifying each connection between an action and a condition can create a generalized explanation. EBL/Soar also needs to introduce some additional constraints on variable names in order to produce legal Soar rules. For example, since one goal cannot have more than a single supergoal, allowing multiple variable names for the supergoal leads to superfluous — i.e., unusable — generality, while also possibly leading to legality problems. The *R-rule* (regressed rule) resulting from the combination of regression and these additional variable constraints is shown in Figure 4-(a). Here, the structure remains the same as in the E-rule.

As shown by the divergence in Figure 2, chunking performs a different transformation here. The variabilization step in chunking is performed by examining the backtrace (explanation) instead of the explanation structure. All constants are left alone; they are never replaced by variables. All object identifiers in the instantiations are replaced by variables; and in particular, all occurrences of the same identifiers are replaced by the same variable. Since E-chunks consist of rules rather than instantiations, we model chunking's variabilization step here as a strengthening of constraints on the explanation structure rather than as a weakening of constraints on the explanation. In particular, if a variable is instantiated as a constant, it is replaced by that

constant; and if a variable is instantiated by an identifier, it remains as a variable, though possibly with a name change (all occurrences of variables which are instantiated by the same identifier are replaced by the same variable). For example, the variables in the E-chunk can be constrained as shown in Figure 4-(b). The pseudo-chunk generated by this step is called an *I-chunk* (instantiation-based chunk). One advantage of this form of instantiation-based constraining over regression (in Soar) is that it naturally introduces the required architectural constraints. For example, the value field of the second condition of R1 and the second condition of R5 are bound to the same identifier G1, and are replaced by the same variable. As long as the instantiations reflect the architectural constraints, the I-chunk automatically preserves them.

However, an I-chunk can be *overspecialized* when distinct variables in the original rules accidentally happen to match the same identifier; for example, although variable <pr> in R2 and variable <pr> in R3 (Figure 1) are instantiated by the same identifier Relation-2, and changed to the same variable <p2>, they can correctly be generalized as different variables, as in Figure 4-(a).[3]

With respect to cost, regression doesn't increase the number of tokens. The number of tokens should remain the same, or be reduced by the extra constraints. I-chunk creation also does not increase cost.

3.4 Eliminating intermediate rule firings (⇒ U-rule)

This step unifies the separate rules in a R-chunk into a single rule, called a *U-rule* (unified rule). Figure 5 shows the result of unifying the R-rule in Figure 4-(a) into the corresponding U-rule. Although R1-R5 still have their own identifiable conditions in the U-rule, there are now no intermediate rule firings. The boundaries between the rules are eliminated by removing the intermediate processes of rule firing and WME creation and testing. In lieu of these processes, the instantiations generated by matching the earlier rules in the firing sequence (i.e., the tokens produced by their final conditions) are passed directly to the match of the later rules.[4] In effect, this replaces the intermediate WMEs with the instantiations which created the WMEs. For example, one of R5's conditions receives the instantiations of R2 directly as

[3]Regression maintains relational tests among the variables bound to the constant, where chunking explicitly replace them by constants. The current implementation of EBL/Soar only allows inequality tests, but will be extended to cover the full set of relational tests.

[4]In addition, all variable tests performed on these intermediate WMEs are transformed into "cross-sub-rule" tests that can be performed on variables in the operational (i.e., fringe) conditions.

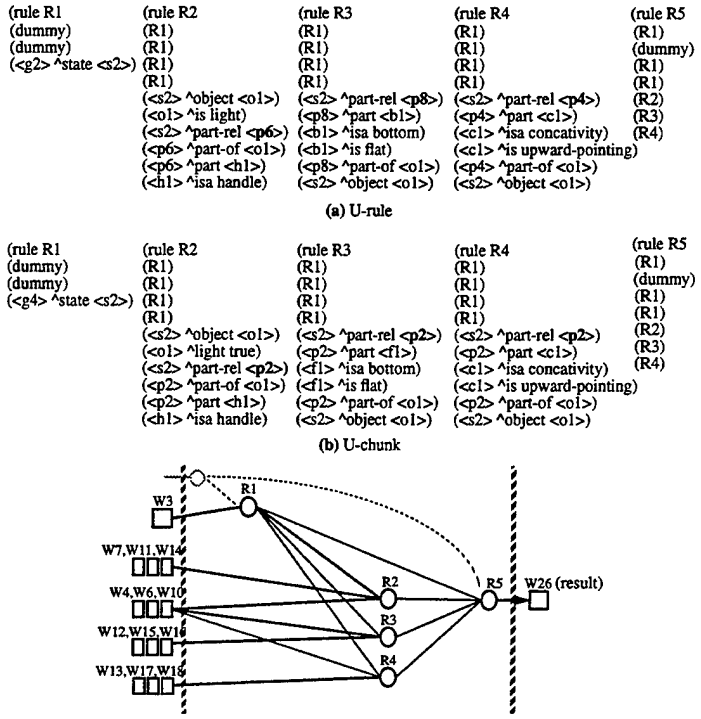


Figure 5: U-rule and U-chunk: created by eliminating intermediate rule firings in the R-rule and I-chunk respectively.

intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1-R5 are no longer (separate) rules.[5] The U-rule corresponds to a *U-chunk*, which is an I-chunk that has been unified as a single rule firing.

Cost problems may be introduced in going to U-rules (and U-chunks), because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations, as explained in [6]. For example, if object O1 has one more handle represented by two more WMEs; (Relation-1 ^part Handle-2) and (Handle-2 ^isa handle), two instantiations of R2 (in Figure 4-(a)) are created instead of one. Because these two instantiations generate the same bindings for variables <s1> and <o1>, only one tuple (WME) is generated in the problem solving (working memory is a set in Soar and other Ops-like languages). In this case, the number of tokens is increased after the WMEs are replaced with the instantiations.

[5]To match U-rules, an extension is required to the Rete algorithm. The traditional form of the algorithm requires a *linear* match network, in the sense that a total ordering must be imposed on the conditions to be matched. However, U-rules require the ability to perform *non-linear* matches, in which conditions are matched hierarchically. They also require the ability to create hierarchically structured tokens.

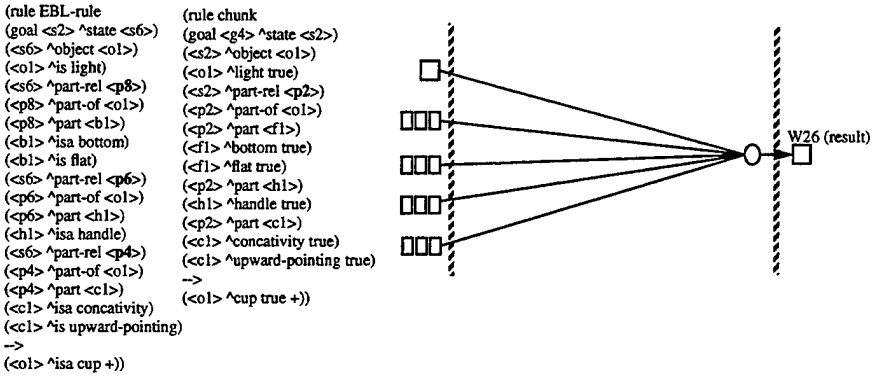


Figure 6: EBL-rule (U-chunk): results from linearizing the U-rule (U-chunk).

The solution previously proposed for this problem in chunking can be adopted here; *preprocessing instantiations before they are used*[6]. By grouping instantiations that generate the same WME, or by selecting one of them as a representative, the number of tokens passed from a substructure of a U-rule may be no greater than the number of WMEs passed in the corresponding R-rule.

3.5 Linearizing ($\Rightarrow$ EBL rule)

A U-rule can be *linearized* to become an EBL-rule. The hierarchical structure of U-rules is flattened into a single layer, and the conditions are totally ordered. For example, the hierarchical structure in Figure 5 can be flattened to the structure in Figure 6. The U-chunk can also be flattened to yield a chunk. After flattening, EBL/Soar and chunking use a heuristic condition-ordering algorithm to further optimize the resulting match.

Unfortunately, linearization turns out to introduce three ways in which match costs can increase. The first way arises directly from the flattening of the U-rule's hierarchical structure. In a U-rule, the conditions in a sub-hierarchy (e.g. the conditions in R1) are matched independently from the other parts of the structure before its created instantiations are joined with the others. By combining these sub-hierarchies together — through linearization — some of the previously independent conditions get joined with other parts of the structure before they finish their sub-hierarchy match. This change can increase the number of tokens. The second way arises from the impact flattening can have on sharing. As long as the implementation of Rete cannot capture the sharing from the hierarchical structure (of the U-rule), the number of tokens can increase. The third way arises because the heuristic condition-ordering algorithm cannot guarantee optimal orderings.

	Number of tokens	
	EBL/Soar	Chunking
Problem Solving	52	52
PS-rule (PS-chunk)	42	42
E-rule (E-chunk)	108	108
R-rule (R-chunk)	108	108
U-rule (U-chunk)	198	198
EBL rule (chunk)	215	215

```
(EBL rule                          (chunk
(goal <g4> ^desired <d1>)          (goal <g4> ^desired <d1>)
(<d1> ^at <l1>)                    (<d1> ^at <l1>)
(<g4> ^operator <o2> +)            (<g4> ^operator <q1> +)
(<o2> ^to <l4>)                    (<q1> ^to <l3>)
(<l4> ^to <l2>)                    (<l3> ^to <l2>)
(<l2> ^to <l1>)                    (<l2> ^to <l1>)
(<l4> ^trans <t2>)                 (<l3> ^trans <t1>)
(<l2> ^trans <t1>)                 (<l2> ^trans <t1>)
-->                                -->
(<g4> ^operator <o2> >))           (<g4> ^operator <q1> >))
```

Figure 7: Results from a grid task.

Whenever this algorithm creates a non-optimal ordering, additional cost may be occurred. The solution to this set of problems may be to *eliminate the linearization step.* By keeping the hierarchical structure — that is, by replacing EBL-rules (or chunks) with U-rules (or U-chunks) — all three causes of cost increase can be avoided. The key thing that this requires is an efficient generalization of Rete for the hierarchical structure of U-rules.

4 Experimental Results

In order to supplement the abstract analysis just provided with experimental evidence, we have implemented a set of learning algorithms that correspond to the set of initial subsequences of the overall transformation sequence; that is, each learning algorithm in the set starts with the problem solving episode and generates a distinct type of (pseudo-) rule or (pseudo-) chunk. At each stage from problem solving to an EBL rule (or chunk), match cost is evaluated by counting the number of tokens required during the match to generate the result.

So far, the resulting experimental system has been applied to a simple grid-task problem[8] which creates one subgoal to break a tie (impasse) among the candidate operators, and creates a search control rule (or chunk). The results of this experiment are shown in Figure 7. The pattern of cost increases matches the expectations generated from the earlier analysis in that transformations led to increased cost on this task if and only if they were identified by the analysis as cost increasing transformations. Moreover, the number of tokens in both system is the same for all (pseudo-) rule and (pseudo-) chunk pairs. However, EBL/Soar creates a more general rule because of the one difference between the two transformational sequences.

5 Summary and Discussion

We have performed an analysis of EBL in Soar as a sequence of transformations from a problem solving episode to a rule. Each step has then been mapped to a corresponding transformation in chunking, and compared in terms of cost and generality. These analyses and comparisons reveal that: (1) the main source of overgeneral learning in Soar stems from the need to use approximate architectural axioms, and is common to EBL and chunking; (2) the main source of overspecial learning in Soar stems from the single transformation that differs between them (chunking does instantiation-based constraining while EBL does regression); (3) chunking automatically incorporates some of Soar's architectural constraints that must be added explicitly with EBL; and (4) the primary sources of expensiveness in Soar's learned rules arise in three transformations that are common between chunking and EBL, and thus might have common solutions. Result (2) shows that EBL and chunking are not all that different. Result (4) goes beyond this to show that the strategies being developed to ensure that chunks are no more costly to use than was the problem solving from which they were learned, should allow a similarly "safe" EBL mechanism to also be developed.

Though the cost analysis is based on Soar (and Rete), the identified causes of expensiveness may crop up in other EBL systems besides EBL/Soar and chunking. First, the cost increase from removal of search control is independent of the match algorithm, and can happen in any system which ignores search control in learning while its problem solving depends on search control. Second, the cost increase from replacing WMEs with instantiations (to unify rules) can occur in any language in which working memory is a set. Finally, the cost increases caused by linearization can occur in any production system that uses a linear match algorithm like Rete. Also, comparable problems may arise in any other system, whenever the original problem solving structure is transformed during the learning process.

Acknowledgments

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan as part of contract N00014-92-K-2015 from the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL).

References

[1] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization – a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[2] G. F. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[3] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325, 1991.

[4] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, 1986. AAAI.

[5] F. Zerr and J. G. Ganascia. Comparison of chunking with EBG implemented onto Soar. Universite Paris-Sud, Orsay, France and Universite Pierre et Marie Curie, Paris, France. October, 1989, Unpublished.

[6] J. Kim and P. S. Rosenbloom. A transformational analysis of expensive chunks. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. IJCAI, 1995. submitted.

[7] M. Tambe, D. Kalp, A. Gupta, C. L. Forgy, B. G. Milnes, and A. Newell. Soar/psm-e: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, 1988.

[8] M. Tambe. *Eliminating combinatorics from production match.* PhD thesis, Carnegie-Mellon University, 1991.

[9] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, 1988.

[10] S. Minton. Personal communication. 1993.

[11] J. E. Laird, P. S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in Soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57, 1986.

[12] J. Kim and P. S. Rosenbloom. Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 174–181, 1993.

[13] R. J. Mooney and S. W. Bennett. A domain independent explanation-based generalizer. In *Proceedings of AAAI-86*, pages 551–555, Philadelphia, 1986.